GLAMkit Smartlinks Documentation

Release 0.5.5

The GLAMkit Association

December 22, 2011

CONTENTS

GLAMkit Eventtools is an open-source event calendar application. It is part of the GLAMkit framework.

This is part of the GLAMkit Project. For more information, please visit http://glamkit.org.

CHAPTER

ONE

GETTING THE CODE

The project is available through Github.

CHAPTER

TWO

CONFIGURATION

1. Add `eventtools` to your `INSTALLED_APPS`
2. Add `(r'^events/', include('events.urls')),` to your urls.py (changing `r'^events/'` to whatever url pattern you'd like glamkit-events to live at)
3. Resync your database `./manage.py syncdb`

2.1 Installation

Download the code; put it into your project's directory or run `python setup.py install` to install system-wide.

REQUIREMENTS: python-vobject (comes with most distribution as a package).

2.2 Settings.py

2.2.1 REQUIRED

***INSTALLED_APPS* - add:** 'events',

***TEMPLATE_CONTEXT_PROCESSORS* - add:** "django.core.context_processors.request",

2.2.2 Optional

FIRST_DAY_OF_WEEK

This setting determines which day of the week your calendar begins on if your locale doesn't already set it. Default is 0, which is Sunday.

OCCURRENCE_CANCEL_REDIRECT

This setting controls the behaviour of `Views.get_next_url()`. If set, all calendar modifications will redirect here (unless there is a *next* set in the request.)

SHOW_CANCELLED_OCCURRENCES

This setting controls the behaviour of `Period.classify_occurence()`. If True, then occurrences that have been cancelled will be displayed with a CSS class of cancelled, otherwise they won't appear at all.

Defaults to False

CHAPTER

THREE

GLAMKIT-EVENTS OVERVIEW

Different institutions have event calendars of differing complexity. GLAMkit-events attempts to cover all the possible scenarios. Before developing with GLAMkit-events, you should spend some time determining what sort of schedule structure you need to model.

3.1 Events, Occurrences and OccurrenceGenerators

GLAMkit-events draws a distinction between **events**, and **occurrences** of those events. **Events** contain all the scheduling information *except* for the times and dates. **Events** know where, why and how things happen, but not when. **OccurrenceGenerators** contain all the *when* information. By combining the two, you can specify individual occurrences of an event. The best way to grasp this is with an example:

> Imagine a museum that has a tour for the blind every Sunday at 2pm. The tour always starts at the same place, costs the same amount etc. The only thing that changes is the date. You can define an event model which has field for storing all the non-time information. By making this model subclass EventBase, you get access to an OccurrenceGenerator which you can use to specify that the tour happens every Sunday at 2pm, and an Occurrence model which handles each specific instance of the tour. This three model structure is handled transparently - you only need to define the event model.

This separation into three models allows us to do some very cool things:

- we can specify complex repetition rules (eg. every Sunday at 2pm, unless it happens to be Easter Sunday, or Christmas day);
- we can attach multiple repetition rules to the same event (eg. the same tour might also happen at 11am every weekday, except during December and January);
- we can specify an end date for these repetition rules, or have them repeat infinitely;
- we can cancel or vary the timing of any specific occurrence irrespective of the underlying rules (eg. on Sunday January 17, 2010, the tour starts two hours earlier);
- we can store special one-off information with any occurrence (eg. on Sunday January 24, 2010, the tour includes lunch);
- we can access all this complexity through an intuitive Django admin interface.

Of course we can also handle ‘one-time’ events. These are simply events with an OccurrenceGenerator that only generates one occurrence

3.2 Different event structures

Let's unpack this further by looking at some of the specific event structures that Glamkit-events can model.

3.2.1 One-off Events

In this paradigm, each event occurs only once.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
9am	A						
10am							
11am							
noon		C					G
1pm				D			
2pm							
3pm					E		
4pm	B						
5pm					F		

In the table above, events A to G each have their own database record, with their own names, descriptions, durations, and any other data that belongs to each event. You might conceivably have relationships with other tables (eg. categories, locations, prices etc.), but the temporal structure of your schedule is very straightforward, and can be modelled with a single database table (and hence a single Django model).

If these are the only seven events you have entered, then next week's schedule will be empty.

If this is how your event schedule works, you're lucky, you can build a workable schedule with a few simple Django models. Glamkit provides an EventBase model that provides a few useful methods, as well as some handy templatetags and admin tweaks.

3.2.2 Simple Recurring Events

Now let's add the concepts of recurring events.

This is where Glamkit-events starts to hit its stride. As well as having a start and end time, events can also have rules that define how the event recurs over time.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
9am	A							A						
10am														
11am	B	B	B	B	B			B	B	B	B	B		
noon														
1pm				D										
2pm	C		C		C		C		C		C		C	
3pm														
4pm														
5pm														

The schedule above could be achieved as follows: Event A:

3.2.3 Complex Recurring Events

	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
9am	A							A						
10am														
11am	B	B	B	B	B			B	B	B	B	B		
noon														
1pm				D										
2pm	B		B		B		B		B		B		B	
3pm														
4pm														
5pm														

3.2.4 Variable Occurrences

	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
9am	A							A						
10am														
11am	B	B	B	B	B			B	B	B	B	B		
noon														
1pm				D										
2pm	C		C		C		C				C		C	
3pm									C					
4pm														
5pm														

3.2.5 Cascading Occurrences

	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
9am	A							A						
10am														
11am	B	B	B	B	B			B	B†	B	B	B		
noon														
1pm				D										
2pm	C		C		C		C				C		C	
3pm									C*					
4pm														
5pm														

3.3 Exceptional Occurrences

Occurrences are generated programatically. This is because we cannot store all of the occurrences in the database, because there could be infinite occurrences. But we still want to be able to vary data about occurrences. Like, cancelling an occurrence, moving an occurrence to a different time or date, or storing a list of attendees with the occurrence.

Exceptional, or Varied Occurrences are saved lazily. An occurrence is generated programatically until it needs to be varied, when saved to the database. When you use any function to get an occurrence, it will be completely transparent whether it was generated programatically or whether it is retrieved from the database (except that retrieved ones will have a `pk` and generated ones don't). Just treat any occurrence like it could be varied and you shouldn't run into any trouble.

CHAPTER FOUR

PERIODS

One of the goals of GLAMkit Events is to make it easy to create events that repeat regularly throughout some period, with exceptions and variation to those repetitions. To do this it creates simple classes for accessing these occurrences. These are Periods. Period is an object that is initiated with an iterable object of events, a start date and start time, and an end date and end time.

It is common to subclass Period for common periods of time. Some of these already exist in the project. eg. Year, Month, Week, Day.

4.1 Period Base Class

This is the base class from which all other periods inherit. It contains all of the base functionality for retrieving occurrences. It is instantiated with a list of events, a start date, and an end date. *The start date is inclusive, the end date is exclusive.* ie [start, end)

```
>>> p = Period(datetime.datetime(2008,4,1,0,0))
```

4.1.1 `get_occurrences()`

This method is for getting the occurrences from the list of passed in events. It returns the occurrences that exist in the period for every event. If I have a list of events `my_events`, and I want to know all of the occurrences from today to next week I simply create a Period object and then call get_occurrences. It will return a sorted list of Occurrences.

```
import datetime

today = datetime.datetime.now()
this_week = Period(my_events, today, today+datetime.timedelta(days=7))
this_week.get_occurrences()
```

4.1.2 `classify_occurrence(occurrence)`

You use this method to determine how the Occurrence `occurrence` relates to the period. This method returns a dictionary. The keys are `"class"` and `"occurrence"`. The class key returns a a number from 0 to 3 and the occurrence key returns the occurrence.

Classes:

0. Only started during this period.
1. Started and ended during this period.

2. Didn’t start or end in this period, but does exist during this period.
3. Only ended during this period.

4.1.3 `get_occurrence_partials()`

This method is used for getting all the occurrences, but getting them as classified occurrences. It simply runs classify_occurrence on each occurrence in get_occurrence and returns that list.

```
import datetime

today = datetime.datetime.now()
this_week = Period(my_events, today, today+datetime.timedelta(days=7))
this_week.get_occurrences() == [classified_occurrence['occurrence'] for \
    classified_occurrence in this_week.get_occurrence_partials()]
```

4.1.4 `has_occurrence()`

This method returns whether there are any occurrences in this period

4.2 Year

The year period is instantiated with a list of events and a date or datetime object. It models the year in which that date exists.

```
>>> p = Year(events, datetime.datetime(2008,4,1))
>>> p.start
datetime.datetime(2008, 1, 1, 0, 0)
>>> p.end
datetime.datetime(2009, 1, 1, 0, 0)
>>> -Remember start is inclusive and end is exclusive
```

4.2.1 `get_months()`

This function returns twelve Month objects which model the twelve months in the Year period.

4.3 Month

The Month period is instantiated with a list of events and a date or datetime object. It models the month that contains the date or datetime object that was passed in.

```
>>> p = Month(events, datetime.datetime(2008,4,4))
>>> p.start
datetime.datetime(2008, 4, 1, 0, 0)
>>> p.end
datetime.datetime(2008, 5, 1, 0, 0)
```

Remember start is inclusive and end is exclusive.

4.3.1 `get_weeks()`

This method returns a list of Week objects that occur at all during that month. The week does not have to be fully encapsulated in the month just have exist in the month at all

4.3.2 `get_days()`

This method returns a list of Day objects that occur during the month.

4.3.3 `get_day(day_number)`

This method returns a specific day in a year given its day number.

4.4 Week

The Week period is instantiated with a list of events and a date or datetime object. It models the week that contains the date or datetime object that was passed in.

```
>>> p = Week(events, datetime.datetime(2008,4,1))
>>> p.start
datetime.datetime(2008, 3, 30, 0, 0)
>>> p.end
datetime.datetime(2008, 4, 6, 0, 0)
>>> -Remember start is inclusive and end is exclusive
```

4.4.1 `get_days()`

This method returns the 7 Day objects that represent the days in a Week period.

4.5 Day

The Day period is instantiated with a list of events and a date or datetime object. It models the day that contains the date or datetime object that was passed in.

```
>>> p = Day(events, datetime.datetime(2008,4,1))
>>> p.start
datetime.datetime(2008, 4, 1, 0, 0)
>>> p.end
datetime.datetime(2008, 4, 2, 0, 0)
>>> -Remember start is inclusive and end is exclusive
```

CHAPTER

FIVE

UTILITIES

There are some utility classes found in the utils module that help with certain tasks.

5.1 EventListManager

EventListManager objects are instantiated with a list of events. That list of events dictates the following methods

5.1.1 `occurrences_after(after)`

Creates a generator that produces the next occurrence inclusively after the datetime `after`.

5.2 OccurrenceReplacer

If you get more into the internals of Glamkit-events, and decide to create your own method for producing occurrences, instead of using one of the public facing methods for this, you are going to want to replace the default generated occurrence with an exceptional one, if an exceptional occurrence exists in the database. To facilitate this in a standardised way you have the OccurrenceReplacer class.

To instantiate it you give it the pool of exceptional occurrences you would like to check in.

```
>>> exceptional_occurrences = my_event.occurrence_set.all()
>>> occ_replacer = OccurrenceReplacer(exceptional_occurrences)
```

Now you have two convenient methods:

5.2.1 `get_occurrence(occurrence)`

This method returns either the passed in occurrence or the equivalent exceptional occurrences from the pool of exceptional occurrences with which this OccurrenceReplacer was instantiated.

```
>>> # my_generated_occurrence is an occurrence that was programatically
>>> # generated from an event
>>> occurrence = occ_replacer.get_occurrence(my_generated_occurrence)
```

5.2.2 `has_occurrence(occurrence)`

This method returns a boolean. It returns True if the OccurrenceReplacer has an occurrence it would like to replace with the give occurrence, and false if it does not.

```
>>> hasattr(my_generated_occurrence, 'pk')
False
>>> occ_replacer.has_occurrence(my_generated_occurrence)
True
>>> occurrence = occ_replacer.get_occurrence(my_generated_occurrence)
>>> hasattr(occurrence, 'pk')
True
>>> # Now with my_other_occurrence which does not have a exceptional counterpart
>>> hasattr(my_other_occurrence, 'pk')
False
>>> occ_replacer.has_occurrence(my_other_occurrence)
False
>>> occurrence = occ_replacer.get_occurrence(my_other_occurrence)
>>> hasattr(occurrence, 'pk')
False
```

CHAPTER

SIX

USEFUL TEMPLATE TAGS

All of the templatetags are located in templatetags/eventstags.py. You can look at more of them there. I am only going to talk about a few here.

To load the template tags your template must contain:

```
{% load eventstags %}
```

6.1 `querystring_for_date`

Usage `{% querystring_for_date <date>[ <num>] %}`

This template tag produces a querystring that describes `date`. It turns date into a dictionary and then turns that dictionary into a querystring, in this fashion:

```
>>> date = datetime.datetime(2009,4,1)
>>> querystring_for_date(date)
'?year=2009&month=4&day=1&hour=0&minute=0&second=0'
```

This is useful when creating links because the calendar_by_period view uses this to display any date besides `datetime.datetime.now()`. The `num` argument can be used to say how specific you want to be about the date. If you were displaying a yearly calendar you only care about the year so `num` would only have to be `1`. See the examples below:

```
>>> querystring_for_date(date, num=1)
'?year=2009'
>>> # Now if we only need the month
>>> querystring_for_date(date, num=2)
'?year=2009&month=4'
>>> # Now everything except the seconds
>>> querystring_for_date(date, num=5)
'?year=2009&month=4&day=1&hour=0&minute=0'
```

CHAPTER

SEVEN

VIEWS

7.1 calendar

This view is for displaying metadata about calendars. Upcoming events, name, description and so on. This is probably not the best view for displaying a calendar in a traditional sense, i.e. displaying a month calendar or a year calendar, as it does not equip the context with any period objects. If you would like to do this you should use calendar_by_period.

7.1.1 Required Arguments

`request` As always the request object.

`calendar_slug` The slug of the calendar to be displayed.

7.1.2 Optional Arguments

`template_name`

> **default** 'events/calendar.html'
>
> This is the template that will be rendered.

7.1.3 Context Variables

`calendar` The Calendar object designated by the `calendar_slug`.

7.2 calendar_by_period

This view is for getting a calendar, but also getting periods with that calendar. Which periods you get is designated with the list `periods`. You can designate which date you want the periods to be initialised to, by passing a date in `request.GET`. See the template tag `query_string_for_date`.

7.2.1 Required Arguments

`request` As always the request object.

`calendar_slug` The slug of the calendar to be displayed.

7.2.2 Optional Arguments

`template_name`

default 'events/calendar_by_period .html'

This is the template that will be rendered.

`periods`

default `[]`

This is a list of period subclasses that designates which periods you would like to instantiate and put in the context.

7.2.3 Context Variables

`date` This was the date that was generated from the query string.

`periods` This is a dictionary that returns the periods from the list you passed in. If you passed in Month and Day, then your dictionary would look like this

```
{
    'month': <events.periods.Month object>
    'day':   <events.periods.Day object>
}
```

So in the template to access the Day period in the context you simply use `periods.day`.

`calendar` This is the Calendar that is designated by the `calendar_slug`.

`weekday_names` This is for convenience. It returns the local names of weekdays for internationalization.

7.3 event

This view is for showing an event. It is important to remember that an event is not an occurrence. Events define a set of recurring occurrences. If you would like to display an occurrence (a single instance of a recurring event) use `occurrence`.

7.3.1 Required Arguments

`request` As always the request object.

`event_id` The id of the event to be displayed.

7.3.2 Optional Arguments

`template_name`

default 'events/calendar_by_period.html'

This is the template that will be rendered.

7.3.3 Context Variables

`event` This is the event designated by the event_id.

`back_url` This is the url that referred to this view.

7.4 occurrence

This view is used to display an occurrence. There are two methods of displaying an occurrence.

7.4.1 Required Arguments

`request` As always the request object.

`event_id` the id of the event that produces the occurrence.

From here you need a way to distinguish the occurrence and that involves.

`occurrence_id` if its exceptional

or it requires a distinguishing datetime as designated by the keywords below. This should designate the original start date of the occurrence that you wish to access. Using `get_absolute_url` from the Occurrence model will help you standardise this.

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`

7.4.2 Optional Arguments

`template_name`

default 'events/calendar_by_period.html'

This is the template that will be rendered.

7.4.3 Context Variables

`event` The event that produces the occurrence.

`occurrence` The occurrence to be displayed.

`back_url` The url from which this request was referred.

7.5 edit_occurrence

This view is used to edit an occurrence.

7.5.1 Required Arguments

`request` As always the request object.

`event_id` The id for the event.

From here you need a way to distinguish the occurrence and that involves

`occurrence_id` The id of the occurrence if its exceptional.

or it requires a distinguishing datetime as designated by the keywords below. This should designate the original start date of the occurrence that you wish to access. Using `get_edit_url` from the Occurrence model will help you standardise this.

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`

7.5.2 Optional Arguments

`template_name`

default 'events/calendar_by_period.html'

This is the template that will be rendered.

7.5.3 Context Variables

`form` An instance of OccurrenceForm to be displayed.

`occurrence` An instance of the occurrence being modified.

7.6 cancel_occurrence

This view is used to cancel an occurrence. It is worth noting that cancelling an occurrence doesn't stop it from being in occurrence lists or being 'exceptional', it just changes the `cancelled` flag on the instance. It is important to check this flag when listing occurrences.

Also if this view is requested via POST, it will cancel the event and redirect. If this view is accessed via a GET request it will display a confirmation page.

7.6.1 Required Arguments

`request` As always the request object.

from here you need a way to distinguish the occurrence and that involves

`occurrence_id` if it's exceptional,

or it requires a distinguishing datetime as designated by the keywords below. This should designate the original start date of the occurrence that you wish to access. Using get_cancel_url from the Occurrence model will help you standardise this.

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`

7.6.2 Optional Arguments

`template_name`

default 'events/calendar_by_period.html'

This is the template that will be rendered, if this view is accessed via GET.

`next`

default The event detail page of `occurrence.event`.

This is the url to which you wish to be redirected after a successful cancellation.

7.6.3 Context Variables

`occurrence` an instance of the occurrence being modified

7.7 create_or_edit_event

This view is used for creating or editing events. If it receives a GET request or if given an invalid form in a POST request it will render the template, or else it will redirect.

7.7.1 Required Arguments

`request` As always the request object

`calendar_id` This is the calendar id of the event being created or edited.

7.7.2 Optional Arguments

`template_name`

default 'events/calendar_by_period.html'

This is the template that will be rendered

`event_id` if you are editing an event, you need to pass in the id of the event, so that the form can be pre-populated with the correct information and also so save works correctly

`next` The url to redirect to upon successful completion or edition.

7.7.3 Context Variables

`form` An instance of EventForm to be displayed.

`calendar` A Calendar with id=calendar_id.

7.8 delete_event

This view is for deleting events. If the view is accessed via a POST request it will delete the event. If it is accessed via a GET request it will render a template to ask for confirmation.

7.8.1 Required Arguments

`request` As always the request object.

`event_id` The id of the event to be deleted.

7.8.2 Optional Arguments

`template_name`

> **default** 'events/calendar_by_period.html'
>
> This is the template that will be rendered.

`next` The url to redirect to after successful deletion.

`login_required`

> **default** `True`
>
> If you want to require a login before deletion happens you can set that here.

7.8.3 Context Variables

`object` The event object to be deleted

CHAPTER

EIGHT

MODELS

CHAPTER

NINE

SETTINGS

9.1 FIRST_DAY_OF_WEEK

This setting determines which day of the week your calendar begins on if your locale doesn't already set it. Default is 0, which is Sunday.

9.2 OCCURRENCE_CANCEL_REDIRECT

This setting controls the behaviour of `Views.get_next_url()`. If set, all calendar modifications will redirect here (unless there is a *next* set in the request.)

9.3 SHOW_CANCELLED_OCCURRENCES

This setting controls the behaviour of `Period.classify_occurence()`. If True, then occurrences that have been cancelled will be displayed with a CSS class of cancelled, otherwise they won't appear at all.

Defaults to False

9.4 CHECK_PERMISSION_FUNC

This setting controls the callable used to determine if a user has permission to edit an event or occurrence. The callable must take the object and the user and return a boolean.

example:

```
check_edit_permission(ob, user):
    return user.is_authenticated()
```

If ob is None, then the function is checking for permission to add new occurrences.

9.5 GET_EVENTS_FUNC

This setting controls the callable that gets all events for calendar display. The callable must take the request and the calendar and return a *QuerySet* of events. Modifying this setting allows you to pull events from multiple calendars or to filter events based on permissions example:

```
get_events(request, calendar):
    return calendar.event_set.all()
```